Brief Introduction to the Work Themes

Milestone 1

Marco Domingues

07/03/2021, Coimbra

Introduction

This document serves as a brief and simple introduction to the subjects that will be contemplated along the internship where it is situated.
The topics are the following: Data Cubes, Data Warehouse, Inverted Index, Frag Cubing and Id Reduction in Data Cubes. Some examples may be provided.

Index

Data Cubes

Data cubes are multi-dimensional arrays of data that usually store references to information far bigger than the computer's main memory.
A data cube is used to represent data along multiple measures of interest. Data cubes allow fast and complete search of the data from different perspectives.

The idea is to organize a group of objects with some specific information along an array with multiple dimensions, where each dimension represents a variable and is ordered in an arbitrary way.

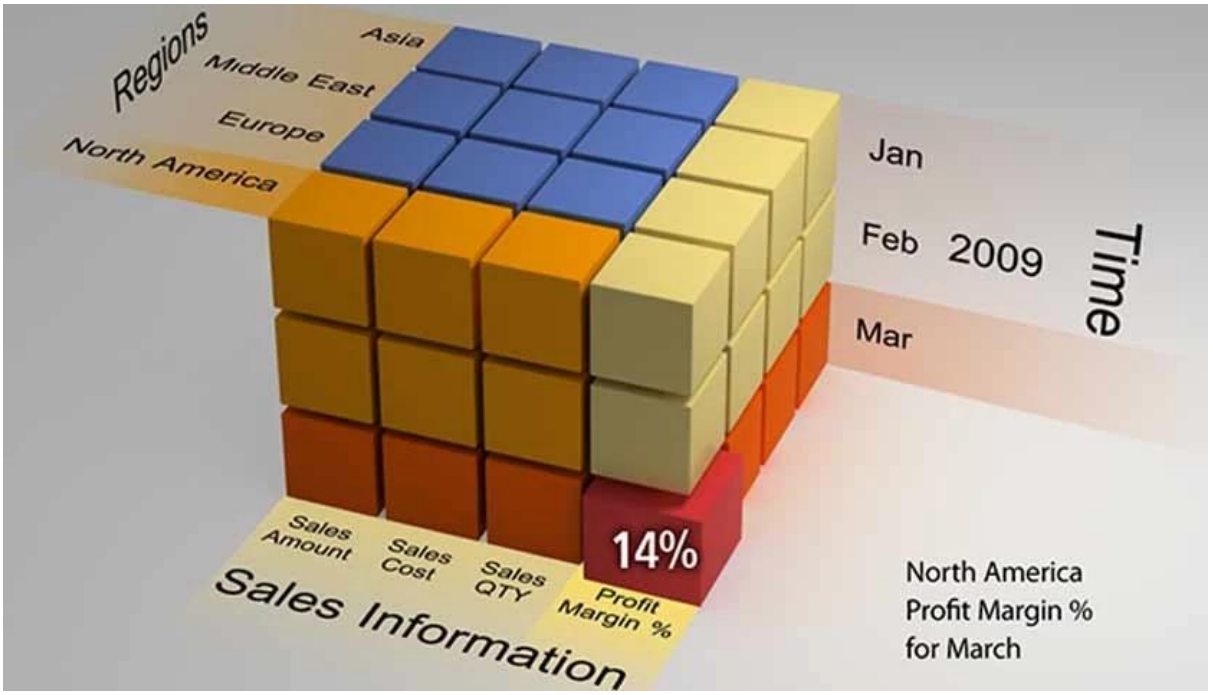

Figure 1: Visual Representation of a Data Cube.

In the image above it is possible to see a visual representation of a data cube with three dimensions. The Z dimension is related to time, therefore, every smaller cube is ordered in a way where the ones with the same time frame are in the same position on the Z axis. Using the same strategy in the other axis results in the creation of an ordered data structure, where searches are fast and easy to do, however it may require a massive amount of memory.

Data Warehouse

Data warehouses are central repositories of data from one or multiple sources (the data stored is just a copy). This system is used both to store data and to allow data analysis. Generally, the data can't be modified, since this kind of system usually works as an archive.

Data warehouses use dimensional models to store the data, since this kind of structure allows for an efficient data organization and retrivery.
Data warehouses have two main approaches to store the data: ETL and ELT.

ETL (Extract, Transform, Load) is a technique where the data is transformed before storing it, that way the database is completely homogeneous, making it easier to search on, the drawback is the difficulty and processing power required to the extraction process due to the diversity and size that the sources may have.

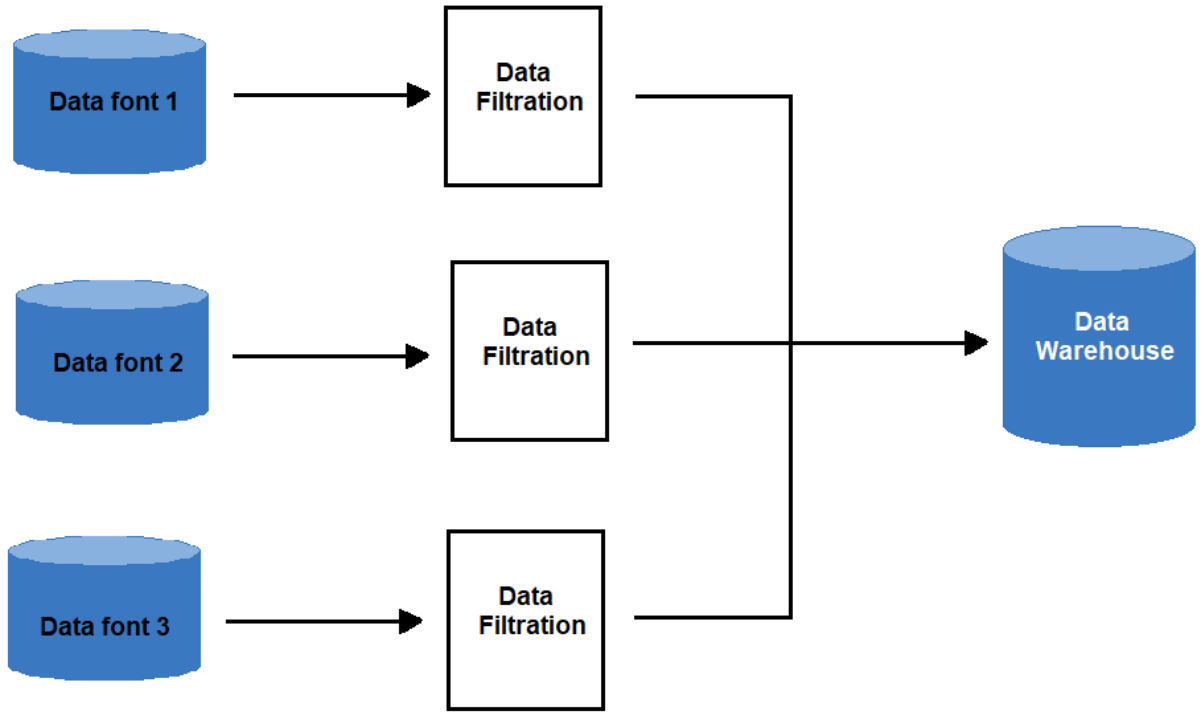

Figure 2: Oversimplified Visual Representation of the ETL process in a Data Warehouse.

ELT (Extract, load, transform) is a technique where the data is stored in the warehouse before any transformation. It has the advantage of not needing to process the data before storing it, however, searches are very time consuming.

Inverted Index

Inverted index is an algorithm developed to map content from a source into a table, where its location is recorded.

A good example is attempting to map words from a text using a table. The table has two columns, one that contains the words and a second one that contains the group of lines where the respective word appears.

This	1,2
is	1,2
a	1
small	1
text	1,2
being	2
used	2
as	2
an	2
example	2

This is a small text.
This text is being used as an example

Figure 3: Visual Representation of an Inverted Index Table.

Frag-Cubing

Data cubes offer fast access to data, however they need a massive amount of memory. If there was no preprocessing done, the amount of time and power needed for each search wouldn't be reliable. Frag-Cubing is an algorithm that attempts to create a middle-ground where there is already some preprocessing done, however it is not so much that it needs the usual massive amount of memory that full data cubes need.
This algorithm is divided into two logical parts, the shell fragments computation and the shell fragment-based query processing.

Shell Fragments Computation

The algorithm behind frag-cubing starts by defining what dimensions go to which shell fragment, this is not a trivial decision and must be made carefully, because having the right dimensions together may increase in a great amount the query processing speed.
The difference between a shell fragment and a full data cube is that the shell fragment only has part of the dimensions, unlike the full data cube that includes all the dimensions of the database. E.G.: If a database's objects have 60 dimensions, while a full data cube would process the entirety of all the dimensions, the shell fragments may have, for example, 3 dimensions each, therefore existing a total of 20 shell fragments.

After defining which dimensions go together in the shell fragments, the algorithm must create an inverted index table with all the dimensions, this is used to make the next steps easier. It must be said that if the information stored in the object dimensions needs more space than the variable type used to represent the TIDs (eg: the dimensions saves words in char arrays where the average word may need 12 bytes to be stored, while the TIDs are stored in an int variable, that, usually, only uses 4 bytes) the inverted index table uses less memory than the database itself.

After the creation of the inverted index table, the frag-cubing algorithm can use any data cube algorithm to process the data into the shell fragments, creating smaller and less memory hungry data cubes, instead of a single massive one.

Shell Fragment-Based Query Processing

Queries

There are two types of queries: point query and subcube query.

A **point query** searches for a specific cuboid cell in the database. The relevant dimensions are instantiated by specific values, while the irrelevant dimensions receive the '*' value. The return of such operation is the inquired value given at the end of the query sentence. A query sentence has the following structure: (<D1>, <D2, …, D<n>: M), where there are 'n' dimensions and the M is the inquired value (the expected return, it may be, for example, an average of a specific value in the database)

A **subcube query** searches for a group of cubes that respect the dimensional values given in the query. A query becomes a subcube query when one of the given dimensions is inquited, which in practical terms means that one of the dimensions must have the value '?'. The return of such a query is essentially a local datacube consisting of the inquired dimensions.

Algorithm

It is proposed a single algorithm that allows both query searches to be satisfied, which is as follows:

For each shell fragment in the database, the algorithm checks if any of the instantiated dimensions is represented in the shell fragment being searched on. In the case where an instantiated dimension is there, the computer must record that dimension in an array used to record all the dimensions. Besides that it must also retrieve the array of TIDs into an array that stores all those arrays. There must be said that a relation between the dimensions retrieved in this process and the array that stored the TIDs of the objects that respect the query is proposed in the original algorithm.
A similar process must be made with the inquired dimensions: The values of the inquired dimensions must be stored in an array and there must be an array that stores the array of the TIDs for the values of the dimensions. There must exist a relation between both arrays, that can be as simple as using the same index in both arrays to the data with the same dimensions.

Following the search loop described above, if one of the arrays that store the TID lists is not empty, then the computer must obtain the list of TIDS that respect all the intanciated dimensions. This process is done simply by intersecting all the values in the TID lists.
In the case where all the TID lists are empty, the algorithm can just return, without doing the above step.
After the above ‘intersection’ step, if there are no inquired dimensions, the algorithm must return the resulting TID list.

If, however, the query has some inquired dimension, the algorithm must take the resulting TID list, the array that stores the inquired dimensions and the array that stores the array with the TIDs of the inquired dimensions to compute a cube based on them and return it.

Proposed changes to the algorithm

The above algorithm, although quite well structured, seems to have room to improve. In this section will be described some process changes that may allow it to improve.
It may be the case where these proportions may be the result of some misunderstanding of the algorithm itself, however, even if that’s the case, they must be taken into account and resolved.

Proposed Change #1:

“In the case where an instantiated dimension is there, the computer must record that dimension in an array used to record all the dimensions. Besides that it must also retrieve the array of TIDs into an array that stores all those arrays. There must be said that a relation between the dimensions retrieved in this process and the array that stored the the TIDs of the objects that respect the query is proposed in the original algorithm”

The algorithm stores the TIDs of the objects that have the value given in an instantiated dimension of a query, which means that the computer, at this point, does the filtration between the objects that respect the instantiated query and those that don’t (for each dimension). With this said, it doesn’t seem that creating an array that stores the dimension values’, which are already known in the query, has any usefulness.

Proposed Change #2:

“Following the search loop described above, if one of the arrays that store the TID lists is not empty, then the computer must obtain the list of TIDS that respect all the intanciated dimensions. This process is done simply by intersecting all the values in the TID lists.”

The immediate continuation condition is “if there is at least one non-null TID list”, which means that the algorithm returns immediately if all the TID lists are empty. Taking into account that the next operation is an interception of all the TID lists it seems that the continuation condition may be improved. For example, imagine that the computer obtained 100 TID lists (which means that the query had 100 instantiated dimensions), if one of those TID lists is empty but the other 99 aren’t, the algorithm will continue to the interception, due to the fact that there is at least one non-null dimension. However, the result of the interception will be null, that happens because the intersection between some universe and a null always returns null (it’s one of the properties of the interception).
With this said, a better continuation condition should be “if there are no empty/ null TID lists”.

Illustrative example:

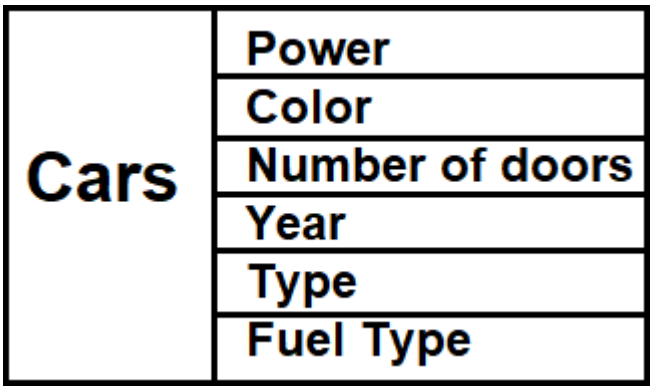

Figure 4: Visual Representation of a Class in a Database.

Using the class Cars represented in the image above, the first step of Frag-cubbing would be deciding how many dimensions the shell fragments should have and how to distribute those dimensions among them would be the first step. A statistical study would be recommended in order to accomplish the best combination of dimensions. Must be emphasized that it is not mandatory for all the shell fragments to have the same number of dimensions, however, for the sake of the example let's divide the six dimensions into two shell fragments, the result must be something close to what is presented in the next picture.

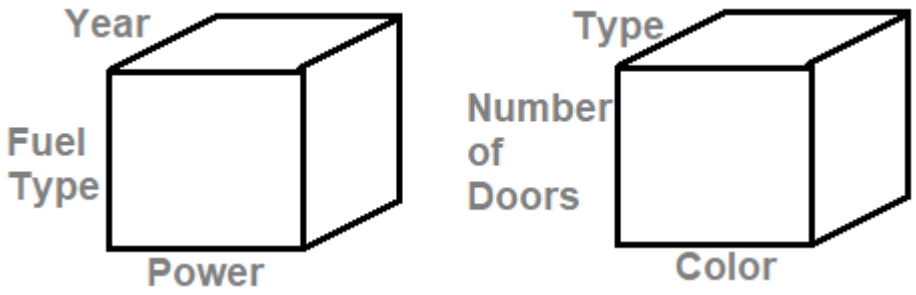

Figure 5: Visual Representation of the shell fragments.

Now that the shell fragments are defined, the next step is to create an inverted index table, that is obtained from the “database” represented by the code in the figure below.

```
new Car( power: 1001, Car.Color.back,  numberDoors: 5,  year: 1972, Car.CarType.city, Car.FuelType.diesel);
new Car( power: 1002, Car.Color.blue,  numberDoors: 3,  year: 1973, Car.CarType.compact, Car.FuelType.electric);
new Car( power: 1003, Car.Color.green,  numberDoors: 3,  year: 1974, Car.CarType.SUV, Car.FuelType.gasoline);
new Car( power: 1001, Car.Color.red,  numberDoors: 3,  year: 1972, Car.CarType.family, Car.FuelType.hybrid);
new Car( power: 1024, Car.Color.white,  numberDoors: 3,  year: 1975, Car.CarType.city, Car.FuelType.gasoline);
new Car( power: 1001, Car.Color.yellow,  numberDoors: 3,  year: 1976, Car.CarType.compact, Car.FuelType.hybrid);
new Car( power: 1003, Car.Color.back,  numberDoors: 5,  year: 1974, Car.CarType.family, Car.FuelType.diesel);
new Car( power: 1021, Car.Color.blue,  numberDoors: 5,  year: 1977, Car.CarType.city, Car.FuelType.hybrid);
new Car( power: 1024, Car.Color.green,  numberDoors: 5,  year: 1974, Car.CarType.SUV, Car.FuelType.electric);
new Car( power: 1032, Car.Color.red,  numberDoors: 5,  year: 1972, Car.CarType.family, Car.FuelType.diesel);
```

Figure 6: Simulation of a Database in Code.

Figure 7: Visual Representation of the Inverted Index Table in Memory.

Type	**City**	**0 4 7**
	SUV	**2 8**
	Family	**3 6 9**
	Compact	**1 5**
Year	**1976**	**5**
	1975	**4**
	1974	**2 6 8**
	1973	**1**
	1972	**0 3 9**
	1977	**7**
Color	**green**	**2 8**
	blue	**1 7**
	white	**4**
	yellow	**5**
	back	**0 6**
Fuel Type	**diesel**	**0 6 9**
	hybrid	**3 5 7**
	electric	**1 8**
	gasoline	**2 4**
Power	**1003**	**2 6**
	1002	**1**
	1024	**4 8**
	1001	**0 3 5**
	1021	**7**
	1032	**9**
Number of Doors	**3**	**1 2 3 4 5**
	5	**0 6 7 8 9**

The inverted index table created in memory must be something among the likes of the table represented in Figure 7.

The next step is to add the data from this table into the shell fragments. That step is not of the concert of frag cubbing, therefore the illustrations will end here.

In order to respond to any queries made, the algorithm just needs to follow the steps indicated above.

ID Reduction in Data Cubes and Update Strategies.

This section is divided in three parts: a way to reduce the number of tuple ids in data cubes, a way to allow the process above to accept changes and a basic design of a possible implementation.

ID Reduction

Given a database with millions of objects (tuples) that store some dimension where, by its nature, the number of different values to that dimension is quite reduced, a few hundreds, for example, it is normal that tuples with TIDs belonging to some interval have the same value. In those cases, it doesn't make sense to store all the TIDs, when it is possible to simply store the TIDs of the tuple in the beginning and the tuple in the end of said interval.

Going back to the table in the figure 7, let's have a closer look to the dimension "Number of Doors":

Number of Doors		
	3	1 2 3 4 5
	5	0 6 7 8 9

Figure 8: A Closer Look into the "Number of Doors" dimension.

In this example, the TIDs of the cars with 3 doors are {1, 2, 3, 4, 5}. Instead of storing these 5 values in memory it is possible to store the two TIDs of each extremity of this interval {1-5}.
For the cars with five doors the same solution may be applied, where, instead of storing {0, 6, 7, 8, 9} the computer can save memory by using the same strategy.

Data Updates

The process above allows data cubes and inverted index tables to store intervals of values, however it brings a new problem: how to deal with changes on tuples that are registered as being part of an interval?

The problem is that any changes may remove a TID from one interval as described above, and create a need to calculate the entire inverted index table and the data cube again. Since those calculations are incredibly heavy another solution needs to be found.

The proposed solution is to create a new table that stores the data of the TID that had changed in each data cube. With this solution, when a change is made, the algorithm must check which data cubes are affected by the changes (changes may not affect all the data of an object) and then add the new values and TID for the changes table that is related to the shell fragment whose dimension was affected.

Let's see an example:
Imagine that dimension D has a measure that stores the interval of IDs from 1-9. Then a change has been made to the tuple with ID of 7, but only the dimension D has been updated. Because the only change is the dimension, the only data cube affected is the one that stores the dimension D (we assume that only one data cube stores the dimension D). In this case, instead of preprocessing the entire cube the algorithm must first, check if the ID 7 is already in the table that stores the changes, T. If the ID 7 is there, then its entry must be updated with the new data. If, however, the ID 7 is not there, a new entry must be made with the id 7, coping all the data from the data cube and updating the dimension D.

To search some information in the data cube, when retrieving the TID lists, the algorithm must also check if there are no changes to the data, by comparing the TIs retrieved with data on the changes table T.

Since this kind of system rarely has updates, it is not expected that this kind of approach can have any meaningful impact either on memory or on performance.